Spark

- Open source system from Berkeley
- Distributed processing over HDFS

- Differences from MapReduce:
  - Multiple steps, including iterations
  - Stores intermediate results in main memory
  - Closer to relational algebra (familiar to you)

- Details: http://spark.apache.org/examples.html

Spark Interface

- Spark supports a Scala interface (and others)
- Scale = extension of Java with functions/closures
- We will illustrate Scala/Spark in the lectures

- Spark also supports a SQL interface, and compiles SQL to its Scala interface

RDD

- RDD = Resilient Distributed Datasets
  - A distributed relation, together with its lineage
  - Lineage: expression that says how that relation was computed

    = a relational algebra plan

- Spark stores intermediate results as RDD

- If a server crashes, its RDD in main memory is lost.

  However, the driver (=master node) knows the lineage, and will simply re-compute the lost partition of the RDD

Programming in Spark

- A Spark/Scala program consists of:
  - Transformations (map, reduce, join…). Lazy
  - Actions (count, reduce, save...). Eager
- RDD[T] = an RDD collection of type T
  - Partitioned, recoverable (through lineage), not nested
- Seq[T] = a Scala sequence
  - Local to a server, may be nested

Example

Given a large log file hdfs://logfile.log retrieve all lines that:

- Start with ERROR
- Contain the string “sqlite”

```
lines = spark.textFile(“hdfs://logfile.log”);

errors = lines.filter(_.startsWith(“ERROR”));

sqlerrors = errors.filter(_.contains(“sqlite”));

sqlerrors.collect()
```

Example

Given a large log file hdfs://logfile.log retrieve all lines that:

- Start with ERROR
- Contain the string “sqlite”

```
lines = spark.textFile(“hdfs://logfile.log”);

errors = lines.filter(_.startsWith(“ERROR”));

sqlerrors = errors.filter(_.contains(“sqlite”));

sqlerrors.collect()
```

Transformation:
Not executed yet…

Action:
triggers execution
of entire program

MapReduce Again…

Steps in Spark resemble MapReduce:

- col.filter(p) applies in parallel the predicate p to all elements x of the partitioned collection, and returns collection with those x where p(x) = true
- col.map(f) applies in parallel the function f to all elements x of the partitioned collection, and returns a new partitioned collection
- Etc

Scala Primer

- Functions with one argument:

```
_.contains("sqlite")
_ > 6
```

- Functions with more arguments

```
(x => x.contains("sqlite"))
(x => x > 6)
((x,y) => x+3*y)
```

- Closures (functions with free variables):

```
var x = 5;   rdd.filter(_ > x)
var s = "sqlite";  rdd.filter(x => x.contains(s))
```

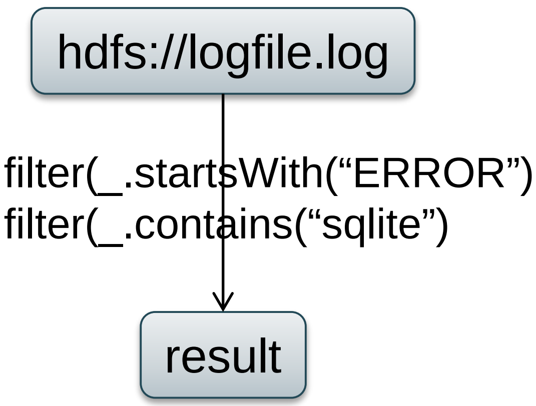
Persistence

```
lines = spark.textFile("hdfs://logfile.log");
errors = lines.filter(_.startsWith("ERROR"));
sqlerrors = errors.filter(_.contains("sqlite"));
sqlerrors.collect()
```

If any server fails before the end, then Spark must restart

Persistence

RDD:

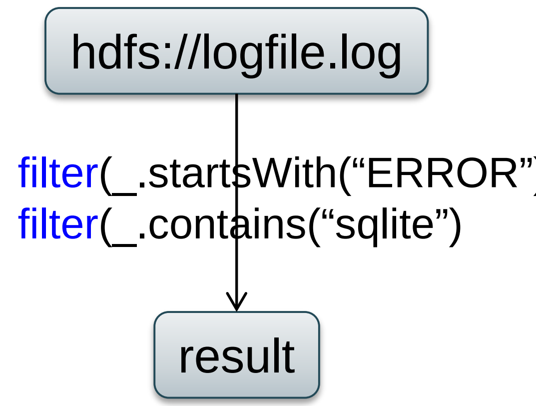



```
lines = spark.textFile(“hdfs://logfile.log”);
errors = lines.filter(_.startsWith(“ERROR”));
sqlerrors = errors.filter(_.contains(“sqlite”));
sqlerrors.collect()
```

If any server fails before the end, then Spark must restart

Persistence

RDD:




```
lines = spark.textFile(“hdfs://logfile.log”);
errors = lines.filter(_.startsWith(“ERROR”));
sqlerrors = errors.filter(_.contains(“sqlite”));
sqlerrors.collect()
```

If any server fails before the end, then Spark must restart

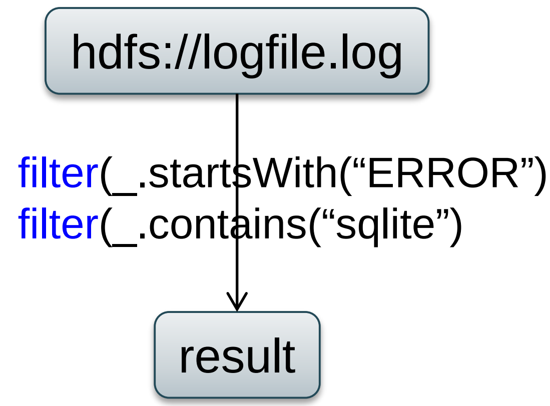
```
lines = spark.textFile(“hdfs://logfile.log”);
errors = lines.filter(_.startsWith(“ERROR”));
errors.persist()
sqlerrors = errors.filter(_.contains(“sqlite”));
sqlerrors.collect()
```

New RDD

Spark can recompute the result from errors

Persistence

RDD:

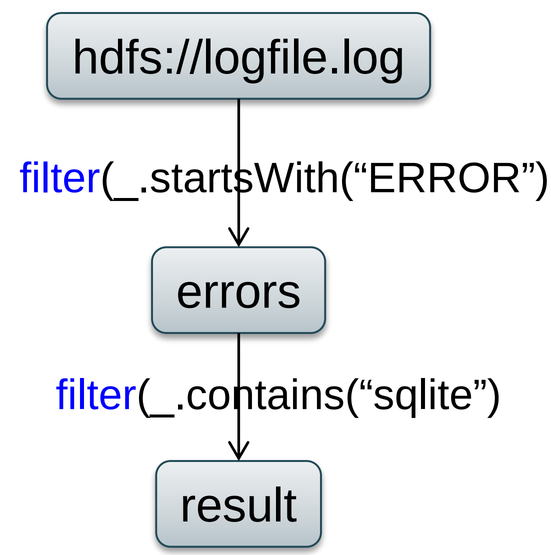



```
lines = spark.textFile(“hdfs://logfile.log”);
errors = lines.filter(_.startsWith(“ERROR”));
sqlerrors = errors.filter(_.contains(“sqlite”));
sqlerrors.collect()
```

If any server fails before the end, then Spark must restart

```
lines = spark.textFile(“hdfs://logfile.log”);
errors = lines.filter(_.startsWith(“ERROR”));
errors.persist()
sqlerrors = errors.filter(_.contains(“sqlite”));
sqlerrors.collect()
```

New RDD



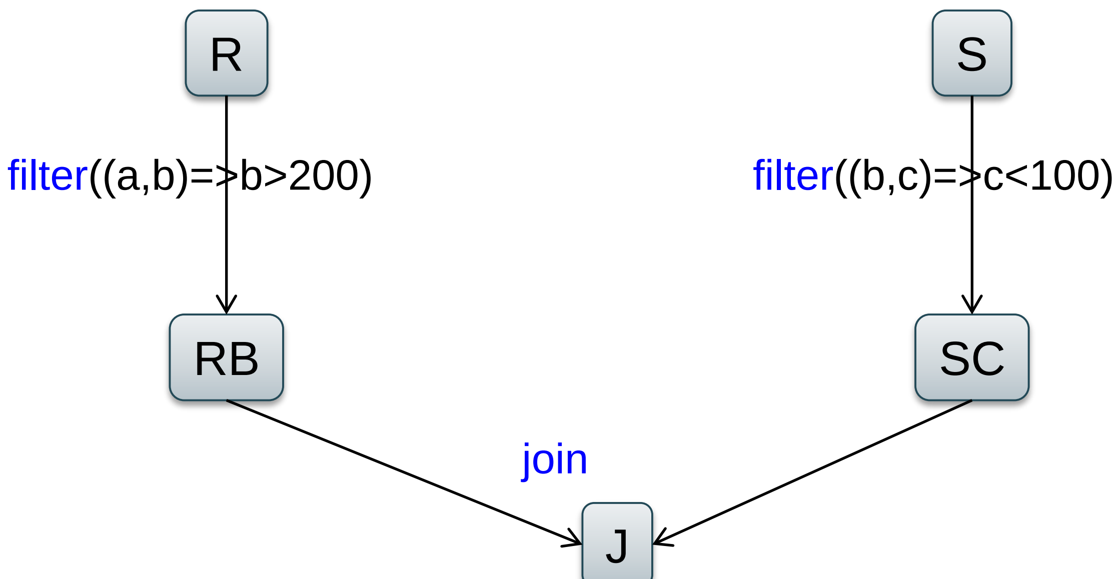
Spark can recompute the result from errors

R(A,B)
S(A,C)

```
SELECT count(*)  FROM R, S
WHERE R.B > 200 and S.C < 100  and R.A = S.A
```

Example

```
R = spark.textFile("R.csv").map(parseRecord).persist()
S = spark.textFile("S.csv").map(parseRecord).persist()
RB = R.filter((a,b) => b > 200).persist()
SC = S.filter((a,c) => c < 100).persist()
J = RB.join(SC).persist
J.count();
```



Programming in Spark

- A Spark/Scala program consists of:
  - Transformations (map, reduce, join…). Lazy
  - Actions (count, reduce, save...). Eager
- RDD[T] = an RDD collection of type T
  - Partitioned, recoverable (through lineage), not nested
- Seq[T] = a Scala sequence
  - Local to a server, may be nested

Transformations:	
map(f : T => U):	RDD[T] => RDD[U]
flatMap(f: T => Seq(U)):	RDD[T] => RDD[U]
filter(f:T=>Bool):	RDD[T] => RDD[T]
groupByKey():	RDD[(K,V)] => RDD[(K,Seq[V])]
reduceByKey(F:(V,V) => V):	RDD[(K,V)] => RDD[(K,V)]
union():	(RDD[T],RDD[T]) => RDD[T]
join():	(RDD[(K,V)],RDD[(K,W)]) => RDD[(K,(V,W))]
cogroup():	(RDD[(K,V)],RDD[(K,W)]) => RDD[(K,(Seq[V],Seq[W]))]
crossProduct():	(RDD[T],RDD[U]) => RDD[(T,U)]

Actions:	
count():	RDD[T] => Long
collect():	RDD[T] => Seq[T]
reduce(f:(T,T)=>T):	RDD[T] => T
save(path:String):	Outputs RDD to a storage system e.g. HDFS

Conclusions

- Parallel databases
  - Predefined relational operators
  - Optimization
  - Transactions
- MapReduce
  - User-defined map and reduce functions
  - Must implement/optimize manually relational ops
  - No updates/transactions
- Spark
  - Predefined relational operators
  - Must optimize manually
  - No updates/transactions